Node.js 是什么

传统意义上的 JavaScript 运行在浏览器上，这是因为浏览器内核实际上分为两个部分:渲染引擎和 JavaScript 引擎。前者负责渲染 HTML + CSS，后者则负责运行 JavaScript。Chrome 使用的 JavaScript 引擎是 V8，它的速度非常快。

Node.js 是一个运行在服务端的框架，它的底层就使用了 V8 引擎。我们知道 Apache + PHP 以及 Java 的 Servlet 都可以用来开发动态网页，Node.js 的作用与他们类似，只不过是使用 JavaScript 来开发。

从定义上介绍完后，举一个简单的例子，新建一个 app.js 文件并输入以下内容:

```
var http = require('http');
http.createServer(function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'}); // HTTP Response 头部
    response.end('Hello World\n'); // 返回数据 “Hello World”
}).listen(8888); // 监听 8888 端口// 终端打印如下信息
console.log('Server running at http://127.0.0.1:8888/');
```

这样，一个简单的 HTTP Server 就算是写完了，输入 node app.js 即可运行，随后访问 便会看到输出结果。

为什么要用 Node.js

面对一个新技术，多问几个为什么总是好的。既然 PHP、Python、Java 都可以用来进行后端开发，为什么还要去学习 Node.js？至少我们应该知道在什么场景下，选择 Node.js 更合适。

总的来说，Node.js 适合以下场景:

1. 实时性应用，比如在线多人协作工具，网页聊天应用等。
2. 以 I/O 为主的高并发应用，比如为客户端提供 API，读取数据库。
3. 流式应用，比如客户端经常上传文件。
4. 前后端分离。

实际上前两者可以归结为一种，即客户端广泛使用长连接，虽然并发数较高，但其中大部分是空闲连接。

Node.js 也有它的局限性，它并不适合 CPU 密集型的任务，比如人工智能方面的计算，视频、图片的处理等。

当然，以上缺点不是信口开河，或者死记硬背，更不是人云亦云，需要我们对 Node.js 的原理有一定的了解，才能做出正确的判断。

基础概念

在介绍 Node.js 之前，理清楚一些基本概念有助于更深入的理解 Node.js 。

并发

与客户端不同，服务端开发者非常关心的一项数据是并发数，也就是这台服务器最多能支持多少个客户端的并发请求。早年的 C10K 问题就是讨论如何利用单台服务器支持 10K 并发数。当然随着软硬件性能的提高，目前 C10K 已经不再是问题，我们开始尝试解决 C10M 问题，即单台服务器如何处理百万级的并发。

在 C10K 提出时，我们还在使用 Apache 服务器，它的工作原理是每当有一个网络请求到达，就 fork 出一个子进程并在子进程中运行 PHP 脚本。执行完脚本后再把结果发回客户端。

这样可以确保不同进程之间互不干扰，即使一个进程出问题也不影响整个服务器，但是缺点也很明显:进程是一个比较重的概念，拥有自己的堆和栈，占用内存较多，一台服务器能运行的进程数量有上限，大约也就在几千左右。

虽然 Apache 后来使用了 FastCGI，但本质上只是一个进程池，它减少了创建进程的开销，但无法有效提高并发数。

Java 的 Servlet 使用了线程池，即每个 Servlet 运行在一个线程上。线程虽然比进程轻量，但也是相对的。有人测试过，每个线程独享的栈的大小是 1M，依然不够高效。除此以外，多线程编程会带来各种麻烦，这一点想必程序员们都深有体会。

如果不使用线程，还有两种解决方案，分别是使用协程(coroutine)和非阻塞 I/O。协程比线程更加轻量，多个协程可以运行在同一个线程中，并由程序员自己负责调度，这种技术在 Go 语言中被广泛使用。而非阻塞 I/O 则被 Node.js 用来处理高并发的场景。

非阻塞 I/O

这里所说的 I/O 可以分为两种: 网络 I/O 和文件 I/O，实际上两者高度类似。I/O 可以分为两个步骤，首先把文件(网络)中的内容拷贝到缓冲区，这个缓冲区

位于操作系统独占的内存区域中。随后再把缓冲区中的内容拷贝到用户程序的内存区域中。

对于阻塞 I/O 来说，从发起读请求，到缓冲区就绪，再到用户进程获取数据，这两个步骤都是阻塞的。

非阻塞 I/O 实际上是向内核轮询，缓冲区是否就绪，如果没有则继续执行其他操作。当缓冲区就绪时，讲缓冲区内容拷贝到用户进程，这一步实际上还是阻塞的。

I/O 多路复用技术是指利用单个线程处理多个网络 I/O，我们常说的 select、epoll 就是用来轮询所有 socket 的函数。比如 Apache 采用了前者，而 Nginx 和 Node.js 使用了后者，区别在于后者效率更高。由于 I/O 多路复用实际上还是单线程的轮询，因此它也是一种非阻塞 I/O 的方案。

异步 I/O 是最理想的 I/O 模型，然而可惜的是真正的异步 I/O 并不存在。Linux 上的 AIO 通过信号和回调来传递数据，但是存在缺陷。现有的 libeio 以及 Windows 上的 IOCP，本质上都是利用线程池与阻塞 I/O 来模拟异步 I/O。

Node.js 线程模型

很多文章都提到 Node.js 是单线程的，然而这样的说法并不严谨，甚至可以说很不负责，因为我们至少会想到以下几个问题:

1. Node.js 在一个线程中如何处理并发请求?
2. Node.js 在一个线程中如何进行文件的异步 I/O?

3. Node.js 如何重复利用服务器上的多个 CPU 的处理能力?

网络 I/O

Node.js 确实可以在单线程中处理大量的并发请求，但这需要一定的编程技巧。我们回顾一下文章开头的代码，执行了 app.js 文件后控制台立刻就会有输出，而在我们访问网页时才会看到“Hello，World”。

这是因为 Node.js 是事件驱动的，也就是说只有网络请求这一事件发生时，它的回调函数才会执行。当有多个请求到来时，他们会排成一个队列，依次等待执行。

这看上去理所当然，然而如果没有深刻认识到 Node.js 运行在单线程上，而且回调函数是同步执行，同时还按照传统的模式来开发程序，就会导致严重的问题。举个简单的例子，这里的“Hello World”字符串可能是其他某个模块的运行结果。假设“Hello World”的生成非常耗时，就会阻塞当前网络请求的回调，导致下一次网络请求也无法被响应。

解决方法很简单，采用异步回调机制即可。我们可以把用来产生输出结果的 response 参数传递给其他模块，并用异步的方式生成输出结果，最后在回调函数中执行真正的输出。这样的好处是，http.createServer 的回调函数不会阻塞，因此不会出现请求无响应的情况。

举个例子，我们改造一下 server 的入口，实际上如果要自己完成路由，大约也是这个思路:

```
var http = require('http');var output = require('./string') // 一个第三方模块
http.createServer(function (request, response) {
    output.output(response); // 调用第三方模块进行输出
}).listen(8888);
```

第三方模块:

```
function sleep(milliSeconds) {  // 模拟卡顿
    var startTime = new Date().getTime();
    while (new Date().getTime() < startTime + milliSeconds);
}
function outputString(response) {
    sleep(10000);  // 阻塞 10s
    response.end('Hello World\n'); // 先执行耗时操作，再输出
}

exports.output = outputString;
```

总之，在利用 Node.js 编程时，任何耗时操作一定要使用异步来完成，避免阻塞当前函数。因为你在为客户端提供服务，而所有代码总是单线程、顺序执行。

如果初学者看到这里还是无法理解，建议阅读“Nodejs 入门”这本书，或者阅读下文关于事件循环的章节。

文件 I/O

异步是为了优化体验，避免卡顿。而真正节省处理时间，利用 CPU 多核性能，还是要靠多线程并行处理。

实际上 Node.js 在底层维护了一个线程池。之前在基础概念部分也提到过，不存在真正的异步文件 I/O，通常是通过线程池来模拟。线程池中默认有四个线程，用来进行文件 I/O。

需要注意的是，我们无法直接操作底层的线程池，实际上也不需要关心它们的存在。线程池的作用仅仅是完成 I/O 操作，而非用来执行 CPU 密集型的操作，比如图像、视频处理，大规模计算等。

如果有少量 CPU 密集型的任务需要处理，我们可以启动多个 Node.js 进程并利用 IPC 机制进行进程间通讯，或者调用外部的 C++/Java 程序。如果有大量 CPU 密集型任务，那只能说明选择 Node.js 是一个错误的决定。

榨干 CPU

到目前为止，我们知道了 Node.js 采用 I/O 多路复用技术，利用单线程处理网络 I/O，利用线程池和少量线程模拟异步文件 I/O。那在一个 32 核 CPU 上，Node.js 的单线程是否显得鸡肋呢？

答案是否定的，我们可以启动多个 Node.js 进程。不同于上一节的是，进程之间不需要通讯，它们各自监听一个端口，同时在最外层利用 Nginx 做负载均衡。

Nginx 负载均衡非常容易实现，只要编辑配置文件即可：

```
http{
    upstream sampleapp {
        // 可选配置项，如 least_conn，ip_hash
        server 127.0.0.1:3000;
        server 127.0.0.1:3001;
        // ... 监听更多端口
    }
    ....
    server{
       listen 80;
       ...
       location / {
          proxy_pass http://sampleapp; // 监听 80 端口，然后转发
```

```
    }
}
```

默认的负载均衡规则是把网络请求依次分配到不同的端口，我们可以用 least_conn 标志把网络请求转发到连接数最少的 Node.js 进程，也可以用 ip_hash 保证同一个 ip 的请求一定由同一个 Node.js 进程处理。

多个 Node.js 进程可以充分发挥多核 CPU 的处理能力，也具有很强大的拓展能力。

事件循环

在 Node.js 中存在一个事件循环(Event Loop)，有过 iOS 开发经验的同学可能会觉得眼熟。没错，它和 Runloop 在一定程度上是类似的。

一次完整的 Event Loop 也可以分为多个阶段(phase)，依次是 poll、check、close callbacks、timers、I/O callbacks 、Idle。

由于 Node.js 是事件驱动的，每个事件的回调函数会被注册到 Event Loop 的不同阶段。比如 fs.readFile 的回调函数被添加到 I/O callbacks，setImmediate 的回调被添加到下一次 Loop 的 poll 阶段结束后，process.nextTick() 的回调被添加到当前 phase 结束后，下一个 phase 开始前。

不同异步方法的回调会在不同的 phase 被执行，掌握这一点很重要，否则就会因为调用顺序问题产生逻辑错误。

Event Loop 不断的循环，每一个阶段内都会同步执行所有在该阶段注册的回调函数。这也正是为什么我在网络 I/O 部分提到，不要在回调函数中调用阻塞方

法，总是用异步的思想来进行耗时操作。一个耗时太久的回调函数可能会让 Event Loop 卡在某个阶段很久，新来的网络请求就无法被及时响应。

由于本文的目的是对 Node.js 有一个初步的，全面的认识。就不详细介绍 Event Loop 的每个阶段了，具体细节可以查看官方文档。

可以看出 Event Loop 还是比较偏底层的，为了方便的使用事件驱动的思想，Node.js 封装了 EventEmitter 这个类:

```
var EventEmitter = require('events');var util = require('util');
function MyThing() {
    EventEmitter.call(this);

    setImmediate(function (self) {
        self.emit('thing1');
    }, this);
    process.nextTick(function (self) {
        self.emit('thing2');
    }, this);
}
util.inherits(MyThing, EventEmitter);
var mt = new MyThing();

mt.on('thing1', function onThing1() {
    console.log("Thing1 emitted");
});

mt.on('thing2', function onThing1() {
    console.log("Thing2 emitted");
});
```

根据输出结果可知，self.emit(thing2) 虽然后定义，但先被执行，这也完全符合 Event Loop 的调用规则。

Node.js 中很多模块都继承自 EventEmitter，比如下一节中提到的 fs.readStream，它用来创建一个可读文件流， 打开文件、读取数据、读取完成时都会抛出相应的事件。

数据流

使用数据流的好处很明显，生活中也有真实写照。举个例子，老师布置了暑假作业，如果学生每天都做一点(作业流)，就可以比较轻松的完成任务。如果积压在一起，到了最后一天，面对堆成小山的作业本，就会感到力不从心。

Server 开发也是这样，假设用户上传 1G 文件，或者读取本地 1G 的文件。如果没有数据流的概念，我们需要开辟 1G 大小的缓冲区，然后在缓冲区满后一次性集中处理。

如果是采用数据流的方式，我们可以定义很小的一块缓冲区，比如大小是 1Mb。当缓冲区满后就执行回调函数，对这一小块数据进行处理，从而避免出现积压。

实际上 request 和 fs 模块的文件读取都是一个可读数据流:

```
var fs = require('fs');var readableStream =
fs.createReadStream('file.txt');var data = '';

readableStream.setEncoding('utf8');// 每次缓冲区满，处理一小块数据
chunk
readableStream.on('data', function(chunk) {
    data+=chunk;
});// 文件流全部读取完成
readableStream.on('end', function() {
    console.log(data);
});
```

利用管道技术，可以把一个流中的内容写入到另一个流中:

```
var fs = require('fs');var readableStream =
fs.createReadStream('file1.txt');var writableStream =
fs.createWriteStream('file2.txt');

readableStream.pipe(writableStream);
```

不同的流还可以串联(Chain)起来，比如读取一个压缩文件，一边读取一边解压，并把解压内容写入到文件中:

```
var fs = require('fs');var zlib = require('zlib');

fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('output.txt'));
```

Node.js 提供了非常简洁的数据流操作，以上就是简单的使用介绍。

总结

对于高并发的长连接，事件驱动模型比线程轻量得多，多个 Node.js 进程配合负载均衡可以方便的进行拓展。因此 Node.js 非常适合为 I/O 密集型应用提供服务。但这种方式的缺陷就是不擅长处理 CPU 密集型任务。

Node.js 中通常以流的方式来描述数据，也对此提供了很好的封装。

Node.js 使用前端语言(JavaScript) 开发，同时也是一个后端服务器，因此为前后端分离提供了一个良好的思路。